大規模データセットと多種ミュータント演算子を利用した自動修正適合性の再計測

前島 葵[1]　肥後 芳樹[1]　柗本 真佑[1]　楠本 真二[1]　安田 和矢[2]

概要：効率的なデバッグ作業の実現を目的とした自動プログラム修正 (APR) に関する研究が数多く行われている．しかしながら，現状の APR 技術で修正できるバグはあまり多くない．このような現状から，「APR 技術がバグを修正しやすいプログラムを人間が書くようになれば，APR 技術により多くのバグを自動で修正できるようになる」との考えにより，自動修正適合性という指標が先行研究により提案された．自動修正適合性とは，対象のプログラムにおいて APR 技術がどの程度バグを修正できそうかを表す指標である．自動修正適合性の利用により，開発者はこの数値が高いソフトウェアを開発できるようになり，その結果として APR 技術が多くのバグを修正できるようになる．先行研究では，自動修正適合性のアイデアが提案される一方，その計測対象が少数のプログラムのみであり，APR 技術とプログラム構造の関係が十分に明らかになったとはいえなかった．本研究では，大規模なプログラム群を対象にして自動修正適合性の計測実験を行う．また，プログラム構造を変化させるためのミューテーション演算子を先行研究よりも多く用いることで，より信頼性の高い数値を算出できるように計測を行う．計測の結果，プログラム構造が異なれば自動修正適合性の値も異なり，プログラムの複雑度や APR ツールによっても自動修正適合性が変化することがわかった．ステートメント数およびサイクロマチック数に対する自動修正適合性の傾向を調査したところ，単純な文の組み合わせで記述されたプログラムほど修正しやすいことがわかった．

1. はじめに

自動プログラム修正 (Automated Program Repair: APR) と呼ばれる技術が注目を集めている [7]．APR とは，バグを含むプログラムからバグを自動的に取り除く技術である．これまでに多くの APR 手法が提案されてはいるが [7]，現在のところ多くのバグについては修正パッチの生成には至っていない．Liu らの研究では，Defects4J [9] に含まれる 395 個のバグのうち 25 個しか正しく修正できなかったと報告された [1]．このような現状から，APR 技術の研究開発に加えて APR の対象であるプログラムの面からも自動修正を支援すべきだと先行研究で提案された [24]．APR 技術にとってバグ修正しやすいプログラムを人間が書くようになれば，APR 技術により多くのバグを自動で修正できるようになるとの考えにより，自動修正適合性という指標が先行研究 [24] により提案された．自動修正適合性とは APR が対象のプログラムに対してどの程度効果的に作用するかを表す指標である．自動修正適合性を利用し対象プログラムを APR が作用しやすいように変更できるようになれば，自動で多くのバグを除去できるようになる．

先行研究ではリファクタリングの前後でプログラムの自動修正適合性が変化することが報告された．しかし，実験の規模が小さく，自動修正適合性とプログラム構造の関係はまだ十分には明らかになってはいない．本研究では，先行研究で提案された自動修正適合性の計測実験を大規模に行うことにより，より信頼性の高い数値を計測することを目指す．具体的には，より多くのプログラムに対してより多くのミューテーション演算子を利用して変異プログラムを生成する．これにより，先行研究に比べて遙かに多い数の変異プログラムから自動修正適合性の数値が計測できるため，どのようなプログラム構造が APR と親和性が高いのかを明らかにできる．

本研究では，同じ機能を持つが構造の異なるプログラムを AtCoder と GitHub から収集し，自動修正適合性の計測を行った．計測の結果，プログラム構造の違いによって自動修正適合性に差が生じることがわかった．また，AtCoder Beginner Contest の問題の難易度が低い解答プログラムほど自動修正適合性が高くなることがわかった．APR ツールによっても自動修正適合性は異なり，`if` 文の条件式に対して修正を試みる APR ツールが AtCoder の解答プログラムと相性が良いことがわかった．さらに，ステートメント数が多いプログラムやサイクロマチック数が大きいプログ

[1] 大阪大学大学院情報科学研究科 大阪府吹田市
[2] 日立製作所 神奈川県横浜市

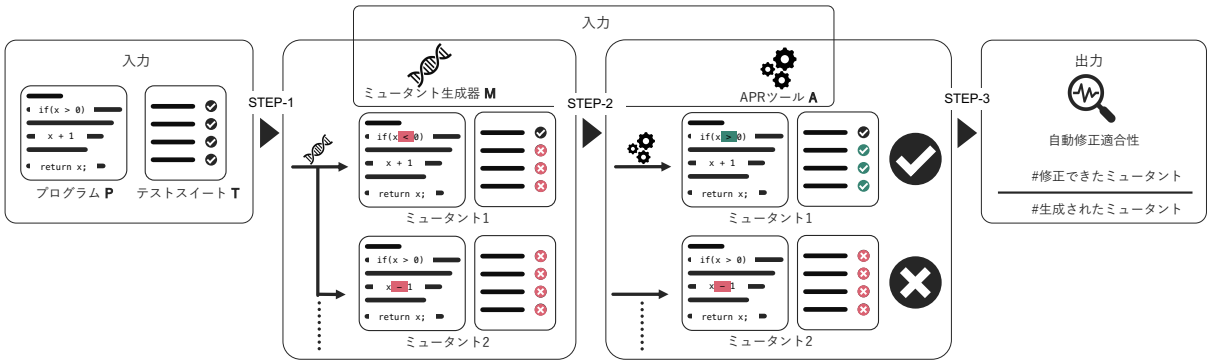


図 1: 自動修正適合性の計測手順

ラムが APR ツールにとって修正しやすいことがわかった．

2. 自動修正適合性

自動修正適合性とは，対象プログラムにおいてバグが発生した場合に，そのバグを APR 技術が修正できる可能性を表現する指標である [24]．先行研究 [24] では，自動修正適合性は下記の状況で利用できると述べられた．

- ソフトウェア開発に APR を導入するのかの判断
- APR による保守を前提としたソフトウェア開発
- APR の効果を高めるためのリファクタリングの研究
- APR が誤ったプログラムを生成する問題の原因究明

自動修正適合性の計測には，以下の 4 項目が必要である．

- プログラム P
- テストスイート T
- ミュータント生成器 M
- APR ツール A

出力はプログラム P の自動修正適合性である．ミュータント生成器によって対象プログラムに適用されるミューテーション演算子が異なるため，生成されるミュータントの数は利用するミュータント生成器によって異なる．同一のバグに対して同一の APR ツールを適用してもテストスイートの品質によって修正の可否が変化する可能性がある [22]．また，APR ツールによって修正の戦略が異なるため修正できるバグも異なる [6]．なお，自動修正適合性の計測は，対象プログラムにおいて顕在化しているバグがない (テストスイート T に含まれる全てのテストケースはプログラム P に対して成功する) ことが前提である．

図 1 は自動修正適合性の計測手順を表している．以下，各手順について説明する．

STEP-1 (ミュータント生成) ミュータント生成器 M を用いてプログラム P からミュータントを複数生成する．各ミュータントは異なるミューテーション演算子の適用により生成されるため，同じミュータントが複数生成されることは無い．

STEP-2 (APR ツールの適用) STEP-1 で生成されたミュータントのうち，テストスイート T に含まれるいずれかのテストケースが失敗するミュータントについては，APR ツール A とテストスイート T を利用して全てのテストケースが成功するプログラムの生成を試みる．生成された状態ですでに全てのテストケースが成功するミュータントについては，APR ツールは実行せず，そのミュータントは STEP-3 においても利用されない．

STEP-3 (自動修正適合性の算出) STEP-2 の実行結果より自動修正適合性を算出する．自動修正適合性は，生成された全ミュータント (全てのテストケースが成功するミュータントを除く) のうち，修正済みプログラムを生成できたミュータントの割合である．例えば，STEP-1 でミュータントが 10 個生成され，STEP-2 で 7 個のミュータントの修正に成功した場合，自動修正適合性は $7/10 = 0.7$ となる．

3. 先行研究の調査結果

先行研究ではリファクタリングを題材に調査が行われ，プログラムの構造によって自動修正適合性に差があることが報告された．その中でも，一時変数のインライン化のリファクタリングは調査が行われた 7 つの APR ツールのうち 5 つのツールで自動修正適合性が向上した．ここでは，先行研究の計測方法について述べ，続く 4 節では本研究の計測方法の変更点について述べる．

3.1 計測対象プログラム

先行研究では，自動修正適合性の調査のために作成された単一のメソッドで構成される小規模なプログラムに対して実験が行われた．機能は同じであるが構造の異なるプログラムを作成するために，リファクタリングを利用した．先行研究では Fowler によって「メソッドの構成」と「条件記述の単純化」に分類されるリファクタリングに限定し，単一のメソッド内で完結する以下の 6 種のリファクタリングが調査対象とした [14]．

*1 先行研究で True Return および False Return とされていたミューテーションは本研究では Change Boolean Literal と統一した．

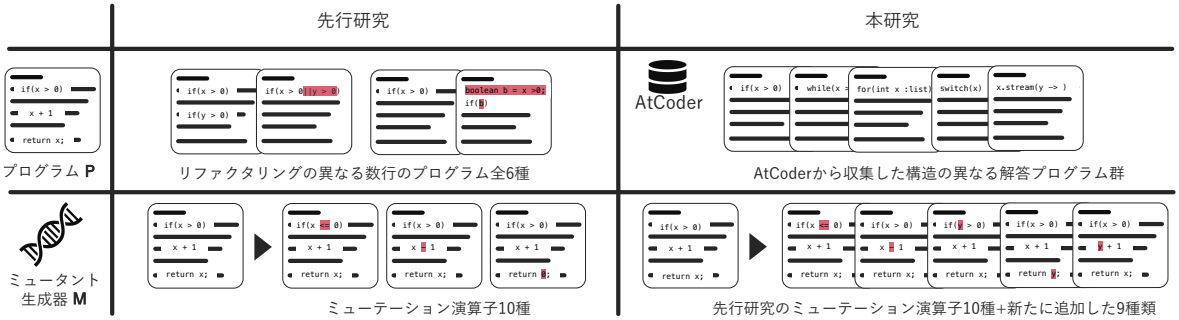


図 2: 先行研究と本研究における計測方法の違い

- 説明用変数の導入
- 一時変数の分離
- 重複した条件記述の断片の統合
- 条件記述の統合
- 制御フラグの削除
- ガード節による入れ子条件記述の置き換え

各リファクタリングの題材につき，リファクタリング適応前と適応後のプログラムが作成された．計測対象プログラム数は 12 である．作成されたプログラムの行数は 9~14 行であり，平均行数は 11 行と小規模であった．

3.2 ミュータント生成器

先行研究では，ミュータントを生成するために PIT [4] のデフォルトミューテーション演算子を利用した (表 1)．PIT はミューテーションテストの分野でミュータントの生成に広く用いられている [21]．

4. 本研究における計測

図 2 に，先行研究と本研究における自動修正適合性の計測方法の違いを示す．4.1 項では本研究での計測対象プログラムについて説明し，4.2 項では本研究で利用したミューテーション演算子について説明する．

表 1: 先行研究で利用されたミューテーション演算子

ミューテーション演算子	変換例	
	変換前	変換後
Conditional Boundary	`a<b`	`a<=b`
Increments	`n++`	`n--`
Invert Negatives	`-n`	`n`
Math	`a+b`	`a-b`
Negate Conditionals	`a==b`	`a!=b`
Void Method Calls	`method();`	`;`
Primitive Returns	`return 5;`	`return 0;`
Empty Returns	`return "str";`	`return "";`
Null Returns	`return object;`	`return null;`
Change Boolean Literal[*1]	`true`	`false`

4.1 計測対象プログラム

本研究では，対象の Java メソッドとして AtCoder [*2] とオープンソースソフトウェアのデータセット [8] を利用した．AtCoder は，AtCoder 社が運営しているプログラミングコンテストサイトである．AtCoder Beginner Contest (ABC) はプログラミング初心者向けのコンテストであり，最も難易度の低い A 問題から最も難易度の高い D 問題の 4 つの問題で構成されている．各問題は問題を説明する問題文，入力の制約，入力・出力の形式およびテストケースに当たる入出力例から構成されている．本研究では，ABC の 101 回から 120 回までの A, B, C, D 問題の解答プログラムを対象とした．各問題に対する解答プログラムの数は，5,546，4,567，2,949，1,247 である．自動修正適合性を計測するためにはテストスイートも必要である．本研究では，ABC において利用されている全テストケース[*3]を利用した．テストケース数の平均値は A 問題が 8.25，B 問題が 11.5，C 問題が 16.75，D 問題が 24.9 である．

また，Java で記述されたオープンソースソフトウェアにおける機能等価メソッドのデータセット [8] も利用した．このデータセットは下記の手順で作成された．

STEP-1 引数の型と返値の型が等しいメソッドは同じ機能を持つ可能性があるとし，同じグループに割り振る．

STEP-2 自動テスト生成ツール Evosuite を利用して，各メソッドからテストケースを生成する．

STEP-3 同じグループ内における全てのメソッドのペアに対して，STEP-2 で生成したテストを相互に実行する．相互テスト実行に成功したメソッドペアを機能等価メソッドペアの候補とする．例えば，メソッド A がメソッド B から生成した全てのテストをパスし，B も A から生成した全てのテストをパスする場合は A と B は機能等価メソッドペアの候補となる．

STEP-4 機能等価メソッドペアの各候補のソースコードを目視で確認し，機能等価であるかを判断する．

以上の手順で収集された 276 の機能等価メソッドグループに属する 728 のメソッドを実験対象ととする．本実験では

[*2] https://atcoder.jp
[*3] `https://atcoder.jp/posts/21`

上記の STEP-1 において Evosuite によって生成されたテストケースも利用した．メソッドあたりのテストケース数の平均値は 3.63 である．

4.2 ミュータント生成器

本研究では，より多くのミュータントを生成するために Simple Stupid Bugs (SStuBs) [10] から得られるバグパターンを用いた．SStuBs は，オープンソース Java プロジェクトから抽出された単一ステートメントのバグ修正のデータセットである．SStuBs では，頻出した単一ステートメントのバグ修正が 16 種類のバグ修正パターンに分類されている．本研究では，バグ修正パターンにおける修正前コードと修正後コードを入れ換えて，16 種類のバグパターンを生成した．さらに，そのうちの 9 種類のバグパターンを用いてソースコードを自動で書き換えるツールを実装した．残りの 7 種類のバグパターンは，すでに先行研究で利用されたミューテーション演算子と同じ，もしくは，そのバグパターンを適用するとコンパイル可能なソースコードが生成できないため，実装しなかった．

4.3 APR ツール

本研究では自動修正適合性の計測に以下の APR ツールを利用した．対象のツールは全て生成と検証に基づく手法の APR ツールである．

jGenProg [23] GenProg [12] の Java 実装版である．遺伝的プログラミングに基づいて修正を行う [17].

jMutRepair [15] Debroy と Wong らによって提案された修正方法であり，条件式および `return` 文のミューテーションによりプログラムを修正する [5].

Cardumen [16] 修正対象のプログラムから取得したテンプレートを用いて式を置換することでプログラムを修正する．jGenProg と同様に文の挿入，削除，置換によって修正を行う．jGenProg は文をそのまま利用するが，Cardumen は文をテンプレートとみなし，文の要素を別の要素に置き換えた文も利用して修正を行う．

jKali [15] コードの削除や `return` 文の挿入によりプログラムを修正する [20].

表 2: 本研究で追加したミューテーション演算子

ミューテーション演算子	変換例	
	変換前	変換後
Change Identifier Used	`Superclass a`	`Subclass a`
Change Numeric Literal	`if(x<0)`	`if(x<1)`
Wrong Function Name	`getEdgeCount()`	`getNodeCount()`
Same Function Less Args	`method(a,b,c)`	`method(a,b)`
Same Function Change Caller	`a.method()`	`b.method()`
Same Function Swap Args	`method(a,b)`	`method(b,a)`
Change Operand	`a+b`	`a+c`
More Specific If	`if(a&&b)`	`if(a)`
Less Specific If	`if(a||b)`	`if(a)`

本研究では，公平に実験を行うため，実行時間制限を 10 秒に統一した．また，jMutRepair は網羅的に探索を行うが，jGenProg, Cardumen, jKali は欠陥限局を用いている．欠陥限局とは，プログラムの欠陥箇所を推測する技術である．欠陥限局は，APR の有効性に影響を与えるといわれる [19]．jGenProg, Cardumen, jKali ではデフォルト設定である GZoltar [3] を利用した．

欠陥限局を行う APR ツールについては，欠陥限局の影響を排除するために，Liu らは欠陥限局が正しく行われた前提で比較を行っている [13]．Liu らの研究は，APR ツールの比較ではなく APR ツールが持つコード修正能力を比較することである．一方，本研究では APR ツールを比較するため，欠陥限局が正しく行われた前提での比較は行わない．

5. AtCoder Beginner Contest に対する実験

この実験は以下の 5 つの Research Question を明らかにするために行った．

RQ1: 先行研究と比較してミュータント数は増加したか？

先行研究の課題であったミュータント数の少なさを本研究では解決できているかを確認する．本研究で新たに追加したミュータント演算子が AtCoder Beginner Contest (ABC) のプログラムに対してどの程度ミュータントを生成できたか確かめる．また，生成されたミュータントのどの程度が修正できたかも調査する．

RQ2: プログラムの複雑さによってどの程度自動修正適合性が異なるか？

ABC には 4 つ難易度がある．難易度が低い問題は単一の `if` 文や `for` 文を用いた簡単な計算処理のみで解答プログラムを記述できる．一方で，難易度が高い問題はソートなどアルゴリズムの知識が必要とされる場合がある．この RQ では，自動修正適合性が問題の難易度によって変化するのかを調査する．

RQ3: APR ツールによって自動修正適合性が異なるか？

この RQ では，4 つの APR ツールを利用し，ツールによって自動修正適合性が変化するのかを調査する．もし自動修正適合性の高い APR ツールが存在すれば，その修正戦略は ABC のプログラムと相性が良いといえる．

RQ4: ステートメント数が少ないほど自動修正しやすいか？

人がプログラムを書く場合には，可読性を高めるためにステートメント数が少ないほど良いとされ，長すぎるメソッドは可読性が低いとされる [2]．APR にとっても人と同様にステートメント数が少ないプログラムほど修正しやすいのか調査を行う．

RQ5: サイクロマチック数が少ないほど自動修正しやすいか？

サイクロマチック数はプログラムの代表的な複雑度メトリクスとして良く用いられる [11], [18]．人がプログラムを書く場合には，サイクロマチック数が少ないほど良いとされる．APR にとっても人と同様にサイクロマチック数が少ないプログラムほど修正しやすいのか調査を行う．

5.1 実験結果と RQ への回答

実験結果から，5 つの RQ に回答する．

RQ1: 先行研究と比較してミュータント数は増加したか？

表 3 にミュータント別の生成数と修正数を示す．最も修正できたミューテーション演算子は NegateConditionals である．これは，境界条件に関するミュータントであり，jMutRepair の修正戦略である `if` 文の条件式の書き換えに合致したことから多く修正できた．

修正できなかったミューテーション演算子は，EmptyReturns と WrongFunctionName であった．EmptyReturns は `return` 文の文字列を `null` にする演算子である．このことから，APR ツールにとって文字列操作は不得意であること

表 3: ミューテーション演算子別の生成数と修正数

(a) 先行研究で利用されたミュータント演算子

ミューテーション演算子	生成数	修正数	修正率 (%)
ConditionalsBoundary	6,280	2,780	44.27%
Increments	2,428	406	16.72%
InvertNegatives	822	121	14.72%
Math	27,725	4,464	16.10%
NegateConditionals	14,876	12,938	86.97%
VoidMethodCalls	6,262	206	3.29%
PrimitiveReturns	23	15	65.22%
EmptyReturns	39	0	0%
NullReturns	0	0	`NaN`
ChangeBooleanLiteral	1,945	866	44.52%
合計	53,281	21,796	40.91%

(b) 本研究で追加したミュータント演算子

ミューテーション演算子	生成数	修正数	修正率 (%)
MoreSpecificIf	1,398	315	22.53%
LessSpecificIf	1,010	50	4.95%
SameFunctionSwapArgs	25	2	8.00%
WrongFunctionName	26	0	0%
ChangeOperand	6,309	1,039	16.47%
ChangeIdentifierUsed	0	0	`NaN`
SameFunctionLessArgs	0	0	`NaN`
SameFunctionChangeCaller	2,671	254	9.51%
ChangeNumericLiteral	41,842	8,269	19.76%
合計	60,400	9,929	16.44%

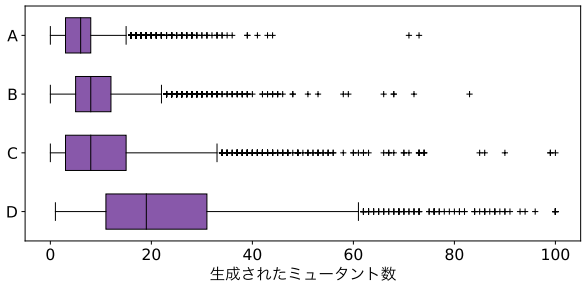


図 3: 各問題から生成されたミュータント数

がわかる．また，WrongFunctionName は変数名を別の変数名に変える演算子であり，APR ツールはメソッド呼び出しに対する修正は他の修正に比べて不得意だといえる．

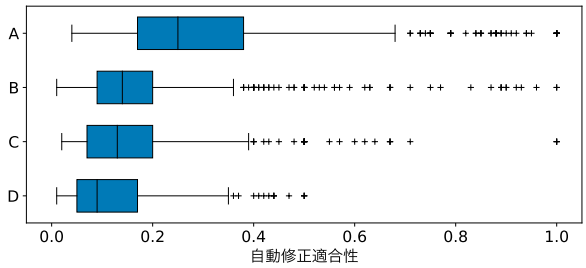
RQ1 への回答： 先行研究で用いられたミューテーション演算子からは 53,281 のミュータントが生成された．本研究で新たに追加した 9 種のミューテーション演算子からは 60,400 のミュータントが生成された．新たにミューテーション演算子を増加することで，ミュータント数を増加できたといえる．また本研究で新たに追加したミューテーション演算子により生成されたミュータントの 0~22.53%が修正できた．

RQ2: プログラムの複雑さによってどの程度自動修正適合性が異なるか？

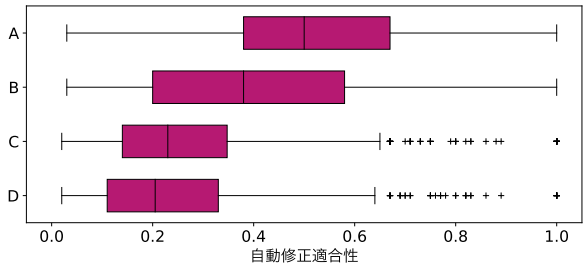
ABC の問題難易度による自動修正適合性の差を調査する．図 3 に，問題の各難易度におけるミュータント生成数を載せる．A 問題はミュータントの生成数が少なく，D 問題は多い．これは，難易度が高くなるほどプログラムが長くなる傾向にあり，ミューテーション演算子が適用できる箇所も多くなるからである．

次に，難易度と自動修正適合性の関係について述べる．図 4 に自動修正適合性の分布を示す．また，各 APR ツールにおいて，A 問題と B 問題，B 問題と C 問題，C 問題と D 問題の自動修正適合性に有意な差があるのかをマンホイットニーの U 検定を用いて調査した．その結果，Cardumen の B 問題と C 問題の間の p 値は 0.018 であったが，それ以外の全ての検定において p 値は 0.01 を下回っていた．これらの結果から，どの APR ツールでも A 問題の自動修正適合性の値が高い傾向にあり，問題の難易度が高くなるに従って自動修正適合性の値が低くなる傾向であることがわかる．

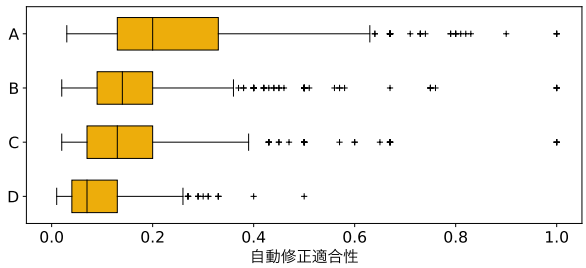
表 3 より，適用されたミューテーション演算子の種類はそのミュータントの修正されやすさに大きく影響することがわかる．また，図 5 は，各難易度の解答プログラムにおいて，ミュータントを生成するために各ミューテーション演算子がどの程度の割合で適用されたのかを表す．この図より，難易度によらず適用されたミューテーション演算



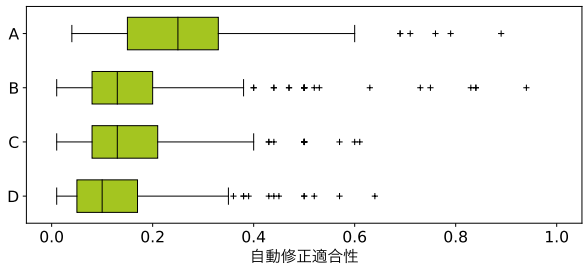
(a) jGenProg



(b) jMutRepair



(c) Cardumen



(d) jKali

図 4: 問題別の自動修正適合性の箱ひげ図 (自動修正適合性 0 の点を除いている)

子の割合にはあまり差がないことがわかる．以上のことから，難易度別のプログラムにおいて，適用されたミューテーション演算子の種類の違いにより自動修正適合性に差が出ているとは考えにくく，難易度が高いほど（プログラムが複雑であるほど），APR によって修正することが難しいといえる．

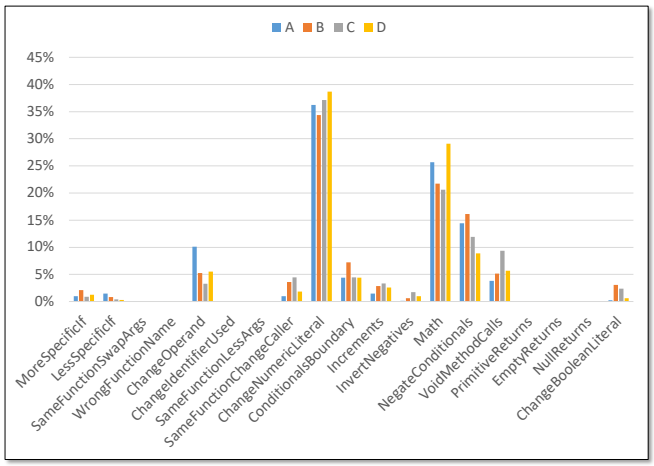


図 5: 各難易度の解答プログラムに対する各ミューテーション演算子の適用割合

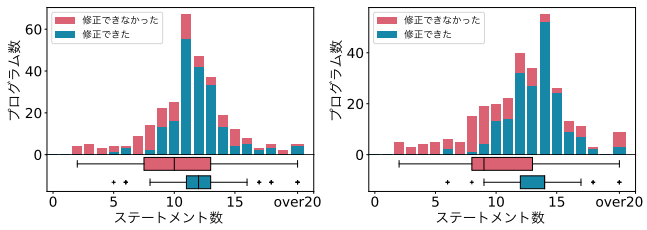
RQ2 への回答: 問題の難易度によって自動修正適合性に差が生じる．難易度の高い問題ほど，自動修正適合性が低くなる傾向があった．

RQ3: APR ツールによって自動修正適合性が異なるか？

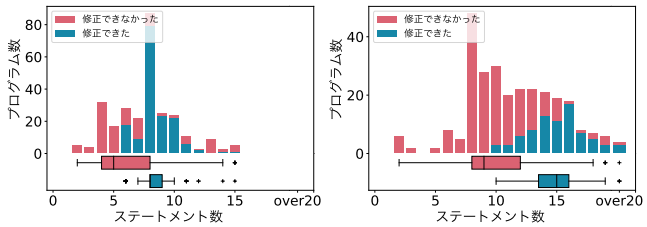
4 つの APR ツールによる自動修正適合性の差を調査する．はじめに，表 4 の右列の APR ツール別の自動修正適合性が 0 より大きいプログラム数に注目する．この列より，自動修正適合性が 0 より大きいプログラムの割合は，jMutRepair, jGenProg,Cardumen, jKali の順に多いことがわかる．図 4 では，各 APR ツールにおいて，問題の難易度別に自動修正適合性の箱ひげ図が並んでいる．どの難易度の問題でも jMutRepair の自動修正適合性の値が高い傾向にあることがわかる．jMutRepair の自動修正適合性が最も大きくなった理由は， 条件式あるいは `return` 文のミューテーションにより修正を行う jMutRepair が最も ABC のプログラムと相性が良いからだと著者らは考えた．ABC は入力された数値に対して数値計算や判定を行う問題

表 4: ABC の解答プログラムに対する自動修正適合性の計測結果

APR ツール	自動修正適合性 (問題別)									
	A 問題		B 問題		C 問題		D 問題		合計	
	0.0	(0.0, 1.0]	0.0	(0.0, 1.0]	0.0	(0.0, 1.0]	0.0	(0.0, 1.0]	0.0	(0.0, 1.0]
jGenProg	4,692	854	3,922	646	2,539	410	1,003	244	12,156	2,154
jMutRepair	3,336	2,210	1,945	2,623	1,851	1,098	687	560	7,819	6,491
Cardumen	4,899	647	3,864	704	2,499	450	977	270	12,239	2,071
jKali	5,203	343	4,289	279	2,681	268	982	265	13,155	1,155
合計	18,130	4,054	14,020	4,252	9,570	2,226	3,649	1,339	45,369	11,861



(a) ABC 101A 問題　(b) ABC 101B 問題



(c) ABC 102A 問題　(d) ABC 102B 問題

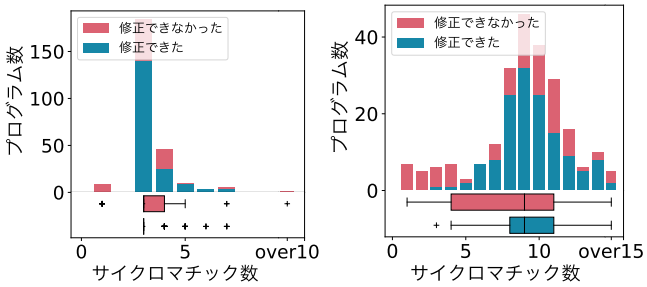
図 6: ステートメント数に対する修正できたプログラムの分布

が多く，分岐が多用されるため条件式が多く登場した．また，文の再利用によって修正を行う戦略を取る jGenProg と Cardumen は修正できるプログラム数も近いという結果になった．jGenProg に対して Cardumen は文をテンプレートとした再利用を行うため jGenProg より優れていると述べられているが [16]，本実験では差が生まれなかった．これは，実験では実行時間を制限してしまったため Cardumen にとって十分な探索が行えなかった可能性がある．いくつかの問題に対して APR ツールの実行制限時間を伸ばして実験したところ，Cardumen によって新たに修正できたミュータントが存在した．また，jKali の戦略である文の削除や `return` 文の挿入は他の戦略に比べてあまり効果がなかったといえる．

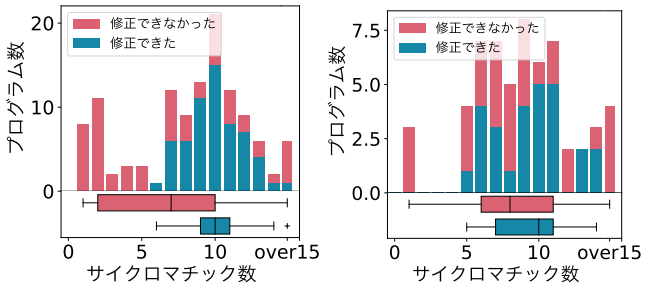
RQ3 への回答: APR ツールによって自動修正適合性に差が生じることがわかった．4 つの APR ツールの中では jMutRepair が自動修正適合性が高い傾向にあり，数値計算や条件判定の多い ABC のプログラムと相性が良いとわかった．自動修正適合性の高い jMutRepair と低い Cardumen では 0.3 程度自動修正適合性に差が生じることがわかった．

RQ4 ： ステートメント数が少ないほど自動修正しやすいか？

図 6 にステートメント数に対する自動修正適合性が 0 より大きかったプログラムの分布を示す．また，この図の下部には修正できなかったミュータントおよび修正できたミュータントのステートメント数の箱ひげ図を表している．この箱ひげ図より，修正できたミュータントにおけるステートメント数の中央値は，修正できなかったミュータントにおけるステートメント数の中央値よりも，大きいことがわかる．ステートメント数が多いほど，1 つの文あたりの複雑度は低下し簡単な演算や分岐でプログラムが構成される．ステートメント数が少ないほど，三項演算子やラムダ式が用いられ分岐が必要とされない．人にとってはステートメント数の少ない方が可読性が高く良いとされるが，APR ツールにとってはステートメント数の多い方が修正しやすい傾向があることがわかった．



(a) ABC 104A 問題　(b) ABC 104B 問題



(c) ABC 104C 問題　(d) ABC 104D 問題

図 7: サイクロマチック数に対する修正できたプログラムの分布

RQ4 への回答: ステートメントが多いプログラムの方が自動修正しやすい．

RQ5 ： サイクロマチック数が少ないほど自動修正しやすいか？

サイクロマチック数の計測には，品質計測ツールの 1 つである PMD [*4]を用いた．図 7 にサイクロマチック数に対する修正できなかったプログラムと修正できたプログラムの分布を示す．修正できなかったプログラムとは自動修正適合性が 0 のプログラムを指し，修正できたプログラムとは自動修正適合性が 0 より大きいプログラムを指す．箱ひげ図に注目すると，修正できなかったプログラムに比べて修正できたプログラムはサイクロマチック数が多い傾向にあることがわかる．

サイクロマチック数でもステートメント数と同様の傾向が得られた．簡単な演算や分岐でプログラムが構成されているほど修正しやすく，三項演算子やラムダ式が用いられ

[*4] https://pmd.github.io/

```
boolean __target__(String s1,String s2){
    if (s1 == null && s2 == null) {
        return true;
    }
    if (s1 == null || s2 == null) {
        return false;
    }
    return s1.equals(s2);
}
```
(a)

```
boolean __target__(String a,String b){
    if ((a == null) && (b == null)) {
        return true;
    }
    else   if ((a == null) || (b == null)) {
        return false;
    }
    else {
        return a.equals(b);
    }
}
```
(b)

```
boolean __target__(String id1,String id2){
    if (id1 == null && id2 == null) {
        return true;
    }
    if (id1 != null && id2 != null) {
        return id1.equals(id2);
    }
    return false;
}
```
(c)

```
boolean __target__(String a,String b){
    if (a == b) {
        return true;
    }
    if (a == null && b != null) {
        return false;
    }
    if (a != null && b == null) {
        return false;
    }
    return a.equals(b);
}
```
(d)

```
boolean __target__(String a,String b){
    if (a == null) {
        return (b == null);
    }
    else {
        return (a.equals(b));
    }
}
```
(e)

```
boolean __target__(String s1,String s2){
    if (s1 == null)   return (s2 == null);
    return s1.equals(s2);
}
```
(f)

```
boolean __target__(String a,String b){
    if (a != null) {
        return a.equals(b);
    }
    return a == b;
}
```
(g)

図 8: 機能等価メソッド

た分岐がないプログラムほど修正しにくい．人にとってはサイクロマチック数の少ないプログラムの方が可読性が高く良いとされるが，APR ツールにとってはサイクロマチック数の多い方が修正しやすいことがわかった．

RQ5 への回答: サイクロマチック数の大きいプログラムの方が自動修正しやすい．

6. OSS データセットに対する実験

この実験では，オープンソースプロジェクトから得られた同機能を持つメソッド群に対して自動修正適合性を計測する．同機能を持つメソッドの構造を比較することにより，どのような構造が修正しやすいかを考察する．

6.1 計測結果

728 の対象メソッドのうち，ミュータントがまったく生成できなかった，もしくは生成されたミュータントが全てのテストを成功してしまい自動修正適合性の計測ができなかったメソッドが 84 存在した．そのため，276 の等価機能メソッドグループのうちの 39 について，自動修正適合性の比較が行えなかった．残りの 237 のメソッドグループについて，自動修正適合性に差が生じたグループおよび差が生じなかったグループの数を表 5 に示す．調査の結果，26%~41% のグループはプログラムの構造によって自動修正適合性が異なることがわかった．

表 5: 自動修正適合性に差が生じたメソッドグループ数

APR ツール	差が生じたグループ数	差が生じなかったグループ数
jGenProg	61 (26%)	176 (74%)
jMutRepair	98 (41%)	139 (59%)
Cardumen	80 (34%)	157 (66%)
jKali	73 (31%)	164 (69%)

6.2 考察

図 8 はこのデータセットに含まれているメソッドを表している．これらのメソッドを用いてどのようなプログラム構造が自動プログラム修正技術に適しているのかを考察する．表 6 はこの図の各メソッドの自動修正適合性の値を表している．

まず，APR ツールに注目する．jGenProg はメソッド (e) のみを修正でき，jMutRepair は全てのプログラムが修正でき，Cardumen は全て修正できず，jKali は 3 つ修正できたという結果になった．ABC の実験結果と同様に jMutRepair が修正できたプログラムが最も多い．これは，ABC と同様に `if` 文による条件式が多用されているプログラムのため，jMutRepair と相性が良かったと考えられる．一方，ABC とは異なり jKali が次に修正できたプログラムが多いという結果になった．これは `if` 文による判定が多用されたプログラム構造が多かったため，`return` 文を挿入する戦略が合致したためだと考えらえる．

次にプログラムの構造に注目する．(a) と (b) は条件式は同じであり，分岐の構造が異なる．具体的には，(a) は `if` 文のみを利用した構造なのに対して，(b) は `else` を利用した構造である．また，最後の `return` 文が `else` ブロックの中にあるかどうかの違いもある．(a) と (b) の自動修正適合性の値を比較すると，jMutRepair は (a) の自動修正適合性が高く，jKali は (b) の自動修正適合性が高い．つまり，jMutRepair は `if` 文を並列した構造の方が修正しやすく，jKali は `if-else` 文と `else` 文を用いた構造の方が修

表 6: 図 8 のプログラムの自動修正適合性

	jGenProg	jMutRepair	Cardumen	jKali
メソッド (a)	0.00	0.57	0.00	0.00
メソッド (b)	0.00	0.29	0.00	0.29
メソッド (c)	0.00	0.43	0.00	0.00
メソッド (d)	0.00	0.71	0.00	0.86
メソッド (e)	1.00	1.00	0.00	0.00
メソッド (f)	0.00	1.00	0.00	0,00
メソッド (g)	0.00	1.00	0.00	0.50

正しやすいことを示唆している．

次に，(a) と (c) を比較する．これらは 2 箇所目の `if` 文の条件式の真偽が逆であり，`return` 文の内容が入れ替わっている．jMutRepair は (a) の方が自動修正適合性がわずかに高いという結果になった．jMutRepair は条件式および `return` 文のミューテーションによりプログラムの修正を試みる．(a) にのみ存在している 2 番目の条件文が (c) にのみ存在している 2 番目の条件文よりも，jMutRepair との相性が良かったと考えられるが，詳細な考察を行うためには更なる実験が必要である．

次に，(a) と (d) を比較する．(a) の 2 箇所目の `if` 文と (d) の 2, 3 箇所目の `if` 文は同じ処理を行っている．(a) の条件を 2 つに分けて記述したものが (d) であるともいえる．(a) と (d) の自動修正適合性を比較すると，jMutRepair は (d) の方がわずかに高いという結果になり，jKali も (d) の方が高いという結果になった．条件はできるだけ細かく分けた方が修正しやすいといえる．

次に, (e) と (f) を比較する．(e) は `else` 文の中に `return` 文が記述されているのに対して，(f) は `else` 文を用いていない．(e) と (f) の自動修正適合性を比較すると，jGenProg は (e) の方が高く，jMutRepair はどちらも 1.0 であった．このことから，jMutRepair は不要な `else` 文の有無によって修正しやすさは変化しないが，jGenProg は `else` 文がある方が（分岐が多い方が）修正しやすいといえる．

次に，(f) と (g) を比較する．これらは 2 箇所目の `if` 文の条件式の真偽が逆になっており，`return` 文の内容が入れ替わっている．(f) と (g) の自動修正適合性を比較すると，jKali は (g) の方が高く，jMutRepair はどちらも 1.0 であった．このことから，jMutRepair は `if` 文の条件式の真偽を入れ替えても修正しやすさは変化しないことがわかる．

7. 本実験の妥当性について留意する点

本研究では 4 種類の限られた APR ツールに対してのみ実験を行っているため，特定のツールの戦略や特性に偏った結果が得られている可能性がある．異なる APR ツールを使用すると，本研究で得られた修正しやすいプログラム構造の結果とは別の構造の方が修正しやすいといった結果が得られる可能性がある．また，本研究では実行時間の短縮のため APR ツールの実験設定である最大実行時間や個体数に制限を設けたため，設定を変更することで違った結果が得られる可能性がある．また，実験のミュータント数を制限するために ABC の解答プログラムにおいて，ミューテーション対象のメソッドを main メソッドと solve メソッドに限定した．他のメソッドに対するミュータントを生成していないため，ミューテーション対象をプログラム全体とした場合と結果が得られる可能性がある．ABC に対する実験ではプログラミングコンテストで公式に利用されているテストケースを利用したため，テストケースの品質が高いと考えられるが，オープンソースソフトウェアに対する実験では Evosuite によって自動生成されたテストケースを利用したため，テストケースの品質が十分に高くない可能性がある．

8. おわりに

本研究では，先行研究で提案された自動修正適合性について調査を行った．自動修正適合性とは APR が対象のプログラムに対してどの程度効果的に作用するかを表す指標である．自動修正適合性の利用により，開発者はこの数値が高いソフトウェアを開発できるようになり，その結果として APR 技術が多くのバグを修正できるようになる．

対象プログラム構造と APR の修正のしやすさの関係をするため，同機能を持つ異なったプログラムを AtCoder の解答プログラムとオープンソースソフトウェアから収集した．また，SStuBs のバグパターンを利用し，ミューテーション演算子を先行研究よりも多く用いることで，より信頼性の高い数値を算出できるように実験を行った．

調査の結果，プログラム構造が異なれば自動修正適合性の値も異なることがわかり，ABC の問題難易度や APR ツールによっても自動修正適合性が変化することがわかった．ステートメント数およびサイクロマチック数に対する自動修正適合性の傾向を調査したところ，ステートメント数が多く，サイクロマチック数が大きいほど自動プログラム修正技術とは親和性が高いことがわかった．ステートメント数が多く，サイクロマチック数が大きいプログラムは，個々のステートメントは単純になっていると予測されるが今回の実験ではそこまでは調査できていない．そのため，更なる実験を行い，ステートメントの単純さと自動修正適合性の関連についても調査を行いたい．

今後の課題として，APR ツールの種類の増加も挙げられる．また，今回の調査ではオーバーフィットを考慮していないため，オーバーフィットを考慮するより厳しい指標の自動修正適合性の算出も挙げられる．

参考文献

[1] Liu , K., Wang , S., Koyuncu , A., Kim, K., Bissyandé, T. F., Kim, D., Wu, P., Klein, J., Mao, X. and Traon, Y. L.: On the Efficiency of Test Suite Based Program Repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs, *Proc. International Conference on Software Engineering*, p. 615–627 (2020).

[2] Buse, R. P. and Weimer, W. R.: Learning a Metric for Code Readability, *IEEE Transactions on Software Engineering*, Vol. 36, No. 4, pp. 546–558 (2010).

[3] Campos, J., Riboira, A., Perez, A. and Abreu, R.: GZoltar: an eclipse plug-in for testing and debugging, *Proc. International Conference on Automated Software Engineering*, pp. 378–381 (2012).

[4] Coles, H., Laurent, T., Henard, C., Papadakis, M. and

Ventresque, A.: PIT: A Practical Mutation Testing Tool for Java (Demo), *Proc. International Symposium on Software Testing and Analysis*, p. 449–452 (2016).

[5] Debroy, V. and Wong, W. E.: Using Mutation to Automatically Suggest Fixes for Faulty Programs, *Proc. International Conference on Software Testing, Verification and Validation*, pp. 65–74 (2010).

[6] Durieux, T., Madeiral, F., Martinez, M. and Abreu, R.: Empirical review of Java program repair tools: a large-scale experiment on 2,141 bugs and 23,551 repair attempts, *Proc. Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 302–313 (2019).

[7] Gazzola, L., Micucci, D. and Mariani, L.: Automatic Software Repair: A Survey, *IEEE Transactions on Software Engineering*, Vol. 45, No. 1, pp. 34–67 (2019).

[8] Higo, Y., Matsumoto, S., Kusumoto, S. and Yasuda, K.: Constructing Dataset of Functionally Equivalent Java Methods Using Automated Test Generation Techniques, *Proc. of the 19th International Conference on Mining Software Repositories* (2022).

[9] Just, R. e., Jalali, D. and Ernst, M. D.: Defects4J : A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs, *Proc. International Symposium on Software Testing and Analysis*, pp. 437–440 (2014).

[10] Karampatsis, R. M. and Sutton, C.: How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset, *Proc. International Conference on Mining Software Repositories*, p. 573–577 (2020).

[11] Lanza, M. and Marinescu, R.: Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems (2006).

[12] Le Goues, C., Nguyen, T., Forrest, S. and Weimer, W.: Genprog: A generic method for automatic software repair, *IEEE Transactions on Software Engineering*, Vol. 38, No. 1, pp. 54–72 (2011).

[13] Liu, K., Koyuncu, A., Bissyandé, T. F., Kim, D., Klein, J. and Le Traon, Y.: You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems, *Proc. International Conference on Software Testing, Verification and Validation*, pp. 102–113 (2019).

[14] Martin, F.: Refactoring: Improving the Design of Existing Code (1999).

[15] Martinez, M. and Monperrus, M.: Astor: A program repair library for java, *Proc. International Symposium on Software Testing and Analysis*, pp. 441–444 (2016).

[16] Martinez, M. and Monperrus, M.: Ultra-large repair search space with automatically mined templates: The cardumen mode of astor, *Proc. International Symposium on Search Based Software Engineering*, pp. 65–86 (2018).

[17] Martinez, M., Weimer, W. and Monperrus, M.: Do the Fix Ingredients Already Exist? An Empirical Inquiry into the Redundancy Assumptions of Program Repair Approaches, *Proc. International Conference on Software Engineering*, p. 492–495 (2014).

[18] McCabe: A Complexity Measure, *Proc. Transactions on Software Engineering*, Vol. 2, pp. 308–320 (1976).

[19] Qi, Y., Mao, X., Lei, Y. and Wang, C.: Using Automated Program Repair for Evaluating the Effectiveness of Fault Localization Techniques, *Proc. International Symposium on Software Testing and Analysis*, p. 191–201 (2013).

[20] Qi, Z., Long, F., Achour, S. and Rinard, M.: An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems, *Proc. International Symposium on Software Testing and Analysis*, ISSTA 2015, p. 24–36 (2015).

[21] Xuan, J. and Monperrus, M.: Test case purification for improving fault localization, *Proc. International Symposium on Foundations of Software Engineering*, pp. 52–63 (2014).

[22] Yi, J., Tan, S. H., Mechtaev, S., Bö hme, M. and Roychoudhury, A.: A Correlation Study between Automated Program Repair and Test-Suite Metrics, *Proc. International Conference on Software Engineering*, p. 24 (2018).

[23] Yuan, Y. and Banzhaf, W.: ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming, *IEEE Transactions on Software Engineering*, Vol. 46, No. 10, pp. 1040–1067 (2020).

[24] 九間哲士, 肥後芳樹, まつ本真佑, 楠本真二, 安田和矢：自動修正適合性：新しいソフトウェア品質指標とその計測, ソフトウェアエンジニアリングシンポジウム, Vol. 2021, pp. 51–58 (2021).